

Type-Based Methods for Termination and Productivity in Coq

Bruno Barras¹ Jorge Luis Sacchini²

¹INRIA Saclay & LIX

²Carnegie Mellon University – Qatar

July 31, 2013

- Coq is a **total** dependently-typed programming language
- **Totality** means:
 - ▶ Functions must be defined in their entire domain (no partial functions)
 - ▶ Recursive functions must be **terminating**
 - ▶ Co-recursive functions must be **productive**
- Non-terminations leads to inconsistencies
Ex: $(\text{let } f\ x = f\ x \text{ in } f\ 0) : 0 = 1$
- Totality ensures logical consistency and decidability of type checking

Coq

- Termination and productivity are undecidable problems
- Approximate the answer
- Coq imposes **syntactic restrictions** on (co-)recursive definitions
- For termination: guarded-by-destructors
- Recursive calls performed only on **structurally smaller terms**

$$\frac{\Gamma(f : I \rightarrow T) \vdash M : I \rightarrow T}{\Gamma \vdash (\text{fix } f : I \rightarrow T := M) : I \rightarrow T}$$

Coq

- Termination and productivity are undecidable problems
- Approximate the answer
- Coq imposes **syntactic restrictions** on (co-)recursive definitions
- For termination: guarded-by-destructors
- Recursive calls performed only on **structurally smaller terms**

$$\frac{\Gamma(f : I \rightarrow T) \vdash M : I \rightarrow T \quad \mathcal{G}(f, M)}{\Gamma \vdash (\text{fix } f : I \rightarrow T := M) : I \rightarrow T}$$

- The predicate $\mathcal{G}(f, M)$ checks that all recursive calls of f in M are guarded by destructors

- Termination and productivity are undecidable problems
- Approximate the answer
- Coq imposes **syntactic restrictions** on (co-)recursive definitions
- For termination: guarded-by-destructors
- Recursive calls performed only on **structurally smaller terms**

$$\frac{\Gamma(f : I \rightarrow T) \vdash M : I \rightarrow T \quad \mathcal{G}(f, M)}{\Gamma \vdash (\text{fix } f : I \rightarrow T := M) : I \rightarrow T}$$

- The predicate $\mathcal{G}(f, M)$ checks that all recursive calls of f in M are guarded by destructors
- Actually, the guard condition is checked on a normal form of the body

$$\frac{\Gamma(f : I \rightarrow T) \vdash M : I \rightarrow T \quad M \rightarrow^* N \quad \mathcal{G}(f, N)}{\Gamma \vdash (\text{fix } f : I \rightarrow T := M) : I \rightarrow T}$$

Termination in Coq

- Typical example:

```
fix half : nat → nat := λx. case x of
  | 0 ⇒ 0
  | S 0 ⇒ 0
  | S (S p) ⇒ S(half p)
```

Recursive call is **guarded**. The recursive argument is smaller.

- The initial implementation of **G** (due to Eduardo Giménez around 1994) has been extended over the years to allow more functions.
- Most recent extension: commutative cuts (due to Pierre Boutillier).

Termination in Coq

- Typical example:

fix **half** : nat → nat := λx. case x of

| 0 ⇒ 0

| S 0 ⇒ 0

| S (S p) ⇒ S(**half** p)

$p < S(S p)$

Recursive call is **guarded**. The recursive argument is smaller.

- The initial implementation of **G** (due to Eduardo Giménez around 1994) has been extended over the years to allow more functions.
- Most recent extension: commutative cuts (due to Pierre Boutillier).

Termination in Coq

Subterm relation

Subtraction:


```
fix minus : nat → nat → nat := λxy. case x, y of
  | 0, _ ⇒ x
  | S x1, 0 ⇒ S x1
  | S x1, S y1 ⇒ minus x1 y1
```


Termination in Coq

Subterm relation

Subtraction:

```
fix minus : nat → nat → nat := λxy. case x, y of
  | 0, _ ⇒ x
  | S x1, 0 ⇒ S x1
  | S x1, S y1 ⇒ minus x1 y1
```




$x_1 \prec x$ (x_1 is a strict subterm of $S x_1 \equiv x$)

Termination in Coq

Subterm relation

Subtraction:

```
fix minus : nat → nat → nat := λxy. case x, y of
  | 0, _ ⇒ x
  | S x1, 0 ⇒ S x1
  | S x1, S y1 ⇒ minus x1 y1
```



$x_1 \prec x$ (x_1 is a strict subterm of $S x_1 \equiv x$)

Division: $\text{div } x \ y = \left\lfloor \frac{x}{y+1} \right\rfloor$


```
fix div : nat → nat → nat := λxy. case x of
  | 0 ⇒ 0
  | S x1 ⇒ S(div (minus x1 y) y)
```

Termination in Coq

Subterm relation

Subtraction:


fix minus : nat → nat → nat := λxy. case x, y of
| 0, _ ⇒ x
| S x₁, 0 ⇒ S x₁
| S x₁, S y₁ ⇒ minus x₁ y₁



$x_1 \prec x$ (x_1 is a strict subterm of $S x_1 \equiv x$)

Division: $\text{div } x \ y = \left\lfloor \frac{x}{y+1} \right\rfloor$

fix div : nat → nat → nat := λxy. case x of
| 0 ⇒ 0
| S x₁ ⇒ S(div (minus x₁ y) y)



minus x₁ y \preceq x₁ \prec S x₁ \equiv x

Termination in Coq

Subterm relation

Subtraction:

```
fix minus : nat → nat → nat := λxy. case x, y of
  | 0, _ ⇒ x
  | S x1, 0 ⇒ S x1
  | S x1, S y1 ⇒ minus x1 y1
```

Division: $\text{div } x \ y = \left\lfloor \frac{x}{y+1} \right\rfloor$


```
fix div : nat → nat → nat := λxy. case x of
  | 0 ⇒ 0
  | S x1 ⇒ S(div (minus x1 y) y)
```

Termination in Coq

Subterm relation

Subtraction:

```
fix minus : nat → nat → nat := λxy. case x, y of
  | 0, _ ⇒ 0
  | S x1, 0 ⇒ S x1
  | S x1, S y1 ⇒ minus x1 y1
```



$x_1 \prec x$ (x_1 is a strict subterm of $S x_1 \equiv x$)

Division: $\text{div } x \ y = \left\lfloor \frac{x}{y+1} \right\rfloor$


```
fix div : nat → nat → nat := λxy. case x of
  | 0 ⇒ 0
  | S x1 ⇒ S(div (minus x1 y) y)
```

Termination in Coq

Subterm relation

Subtraction:


```
fix minus : nat → nat → nat := λxy. case x,y of
  | 0, _ ⇒ 0
  | S x1, 0 ⇒ S x1
  | S x1, S y1 ⇒ minus x1 y1
```



$x_1 \prec x$ (x_1 is a strict subterm of $S x_1 \equiv x$)

Division: $\text{div } x \ y = \left\lfloor \frac{x}{y+1} \right\rfloor$

```
fix div : nat → nat → nat := λxy. case x of
  | 0 ⇒ 0
  | S x1 ⇒ S(div (minus x1 y) y)
```



$\text{minus } x_1 \ y \not\prec x_1 \prec S x_1 \equiv x$

Termination in Coq

Nested fixpoints

Inductive $\text{rose}(A) : \text{Type} := \text{node} : A \rightarrow \text{list}(\text{rose } A) \rightarrow \text{rose } A$

$\text{rmap} := \lambda f : A \rightarrow B. \text{fix } \text{rmap} : \text{rose } A \rightarrow \text{rose } B :=$
 $\lambda t. \text{case } t \text{ of}$
 $\text{node } x \text{ } ts \Rightarrow \text{node } (f \ x) (\text{map } \text{rmap} \ ts)$

$\text{map} := \lambda f : A \rightarrow B. \text{fix } \text{map} : \text{list } A \rightarrow \text{list } B :=$
 $\lambda l. \text{case } l \text{ of}$
 $\text{nil} \Rightarrow \text{nil}$
 $\text{cons } x \text{ } xs \Rightarrow \text{cons } (f \ x) (\text{map } \ xs)$

Termination in Coq

Nested fixpoints

Inductive $\text{rose}(A) : \text{Type} := \text{node} : A \rightarrow \text{list}(\text{rose } A) \rightarrow \text{rose } A$

$\text{rmap} := \lambda f : A \rightarrow B. \text{fix } \text{rmap} : \text{rose } A \rightarrow \text{rose } B :=$
 $\lambda t. \text{case } t \text{ of}$
 $\text{node } x \text{ } ts \Rightarrow \text{node } (f \ x) \ (\text{map } \text{rmap} \ ts)$



$\text{map} := \lambda f : A \rightarrow B. \text{fix } \text{map} : \text{list } A \rightarrow \text{list } B :=$
 $\lambda l. \text{case } l \text{ of}$
 $\text{nil} \Rightarrow \text{nil}$
 $\text{cons } x \text{ } xs \Rightarrow \text{cons } (f \ x) \ (\text{map } xs)$



Termination in Coq

Nested fixpoints

Inductive $\text{rose}(A) : \text{Type} := \text{node} : A \rightarrow \text{list}(\text{rose } A) \rightarrow \text{rose } A$

$\text{rmap} := \lambda f : A \rightarrow B. \text{fix } \text{rmap} : \text{rose } A \rightarrow \text{rose } B :=$
 $\lambda t. \text{case } t \text{ of}$
 $\text{node } x \text{ } ts \Rightarrow \text{node } (f \ x) \ (\text{map } \text{rmap} \ ts)$

$\text{map} := \text{fix } \text{map} : (A \rightarrow B) \rightarrow \text{list } A \rightarrow \text{list } B :=$
 $\lambda f \ l. \text{case } l \text{ of}$
 $\text{nil} \Rightarrow \text{nil}$
 $\text{cons } x \ xs \Rightarrow \text{cons } (f \ x) \ (\text{map } f \ xs)$



Termination in Coq

Nested fixpoints

Inductive $\text{rose}(A) : \text{Type} := \text{node} : A \rightarrow \text{list}(\text{rose } A) \rightarrow \text{rose } A$

$\text{rmap} := \lambda f : A \rightarrow B. \text{fix } \text{rmap} : \text{rose } A \rightarrow \text{rose } B :=$
 $\lambda t. \text{case } t \text{ of}$
 $\text{node } x \text{ } ts \Rightarrow \text{node } (f \ x) \ (\text{map } \text{rmap} \ ts)$



$\text{map} := \text{fix } \text{map} : (A \rightarrow B) \rightarrow \text{list } A \rightarrow \text{list } B :=$
 $\lambda f \ l. \text{case } l \text{ of}$
 $\text{nil} \Rightarrow \text{nil}$
 $\text{cons } x \ xs \Rightarrow \text{cons } (f \ x) \ (\text{map } f \ xs)$



Syntactic criteria

Limitations

- Works on syntax: small changes in code can make functions ill-typed
- Not compositional
- Difficult to understand for users
 - ▶ Many questions about termination in the Coq list
 - ▶ Error messages not informative
- Difficult to implement: termination checking is the most delicate part of Coq's kernel
- Inefficient: guard condition is checked on the normal form of fixpoints bodies
- Difficult to study
 - ▶ Little documentation
 - ▶ Complicated to even define

Termination in Coq

- Many ways to get around the guard condition:
 - ▶ Adding extra argument to act as measure of termination
 - ▶ Wellfounded recursion
 - ▶ Ad-hoc predicate (Bove)
 - ▶ Tool support (Function, Program)
- But this complicates function definition
- May affect efficiency

Termination using sized types

- Long history: Haskell [Pareto et al.], $\lambda^{\hat{}}$ [Joao Frade et al.], $F_{\omega}^{\hat{}}$ [Abel], $CIC^{\hat{}}$ [Barthe et al.], CC+rewriting [Blanqui et al.] ...

Termination using sized types

- Long history: Haskell [Pareto et al.], $\lambda^{\hat{}}$ [Joao Frade et al.], $F_{\omega}^{\hat{}}$ [Abel], $CIC^{\hat{}}$ [Barthe et al.], CC+rewriting [Blanqui et al.] ...
- Basic idea: user-defined datatypes are decorated with size information

$$\text{nat} ::= O : \text{nat} \mid S : \text{nat} \rightarrow \text{nat}$$

Intuitive meaning: $[\text{nat}] = \{O, S O, S(S O), \dots\}$

Termination using sized types

- Long history: Haskell [Pareto et al.], $\lambda^{\hat{}}$ [Joao Frade et al.], $F_{\omega}^{\hat{}}$ [Abel], $CIC^{\hat{}}$ [Barthe et al.], CC+rewriting [Blanqui et al.] ...
- Basic idea: user-defined datatypes are decorated with size information

$$\text{nat} ::= O : \text{nat} \mid S : \text{nat} \rightarrow \text{nat}$$

Intuitive meaning: $[\text{nat}] = \{O, S O, S(S O), \dots\}$

- Sized types are approximations

$$\text{nat}\langle s \rangle$$

Intuitive meaning: $[\text{nat}\langle s \rangle] = \{O, S O, \dots, \underbrace{S(\dots(S O)\dots)}_{s-1}\}$

Termination using sized types

- Size annotations keep track of the size of elements

$$s ::= \iota \mid \widehat{s} \mid \infty$$

Termination using sized types

- Size annotations keep track of the size of

$$\widehat{\infty} = \infty$$

$$s ::= \iota \mid \widehat{s} \mid \infty$$

Termination using sized types

- Size annotations keep track of the size of elements

$$s ::= \iota \mid \widehat{s} \mid \infty$$

$$\frac{}{\Gamma \vdash O : \text{nat}} \qquad \frac{\Gamma \vdash M : \text{nat}}{\Gamma \vdash S M : \text{nat}}$$

Termination using sized types

- Size annotations keep track of the size of elements

$$s ::= \iota \mid \widehat{s} \mid \infty$$

$$\frac{}{\Gamma \vdash O : \text{nat}\langle \widehat{s} \rangle} \qquad \frac{\Gamma \vdash M : \text{nat}\langle s \rangle}{\Gamma \vdash S M : \text{nat}\langle \widehat{s} \rangle}$$

Termination using sized types

- Size annotations keep track of the size of elements

$$s ::= \iota \mid \widehat{s} \mid \infty$$

$$\frac{}{\Gamma \vdash O : \text{nat}\langle \widehat{s} \rangle}$$

$$\frac{\Gamma \vdash M : \text{nat}\langle s \rangle}{\Gamma \vdash S M : \text{nat}\langle \widehat{s} \rangle}$$

upper bound

Termination using sized types

- Size annotations keep track of the size of elements

$$s ::= \iota \mid \widehat{s} \mid \infty$$

$$\frac{}{\Gamma \vdash O : \text{nat}\langle \widehat{s} \rangle} \qquad \frac{\Gamma \vdash M : \text{nat}\langle s \rangle}{\Gamma \vdash S M : \text{nat}\langle \widehat{s} \rangle}$$

- Substage relation

$$\frac{}{s \sqsubseteq \widehat{s}} \qquad \frac{}{s \sqsubseteq \infty}$$

defines a subtype relation

$$\frac{s \sqsubseteq r}{\text{nat}\langle s \rangle \leq \text{nat}\langle r \rangle}$$

Termination using sized types

Fixpoint rule

Recursive functions are defined on approximations of datatypes:

$$\frac{\Gamma(f : I \rightarrow T) \vdash M : I \rightarrow T}{\Gamma \vdash (\text{fix } f : I \rightarrow T := M) : I \rightarrow T}$$

Termination using sized types

Fixpoint rule

Recursive functions are defined on approximations of datatypes:

$$\frac{\Gamma(f : I\langle i \rangle \rightarrow T) \vdash M : I\langle \widehat{i} \rangle \rightarrow T}{\Gamma \vdash (\text{fix } f : I \rightarrow T := M) : I\langle s \rangle \rightarrow T} \quad i \text{ fresh}$$

- Recursive calls on terms of smaller size

Termination using sized types

Fixpoint rule

Recursive functions are defined on approximations of datatypes:

$$\frac{\Gamma(f : I\langle i \rangle \rightarrow T) \vdash M : I\langle \widehat{i} \rangle \rightarrow T}{\Gamma \vdash (\text{fix } f : I \rightarrow T := M) : I\langle s \rangle \rightarrow T} \quad i \text{ fresh}$$

- Recursive calls on terms of smaller size
- Size-preserving functions: return type T can depend on i

Termination using sized types

Fixpoint rule

Recursive functions are defined on approximations of datatypes:

$$\frac{\Gamma(f : I\langle i \rangle \rightarrow T) \vdash M : I\langle \widehat{i} \rangle \rightarrow T}{\Gamma \vdash (\text{fix } f : I \rightarrow T := M) : I\langle s \rangle \rightarrow T} \quad i \text{ fresh}$$

- Recursive calls on terms of smaller size
- Size-preserving functions: return type T can depend on i
- Non-structural recursion

Example: quicksort

Non-structural recursion

$$\begin{aligned} \text{filter} &\equiv \dots : \Pi A. (A \rightarrow \text{bool}) \rightarrow \text{list } A \rightarrow \text{list } A \times \text{list } A \\ (++) &\equiv \dots : \Pi A. \text{list } A \rightarrow \text{list } A \rightarrow \text{list } A \end{aligned}$$

Example: quicksort

Non-structural recursion

$$\text{filter} \equiv \dots : \Pi A. (A \rightarrow \text{bool}) \rightarrow \text{list } A \rightarrow \text{list } A \times \text{list } A$$
$$(\text{++}) \equiv \dots : \Pi A. \text{list } A \rightarrow \text{list } A \rightarrow \text{list } A$$

fix **qsort** : list A → list A :=

λx : list A. case x of

| nil ⇒ nil

| cons h t ⇒ let (s, g) = filter (< h) t in
(**qsort** s) ++ (cons h (**qsort** g))

Example: quicksort

Non-structural recursion

$$\text{filter} \equiv \dots : \Pi A. (A \rightarrow \text{bool}) \rightarrow \text{list}\langle s \rangle A \rightarrow \text{list}\langle s \rangle A \times \text{list}\langle s \rangle A$$
$$(\text{++}) \equiv \dots : \Pi A. \text{list}\langle s \rangle A \rightarrow \text{list}\langle r \rangle A \rightarrow \text{list}\langle \infty \rangle A$$

```
fix qsort : list A → list A :=  
λx : list A. case x of  
  | nil ⇒ nil  
  | cons h t ⇒ let (s, g) = filter (< h) t in  
                (qsort s ) ++ (cons h (qsort g ))
```

Example: quicksort

Non-structural recursion

$$\text{filter} \equiv \dots : \Pi A. (A \rightarrow \text{bool}) \rightarrow \text{list}\langle s \rangle A \rightarrow \text{list}\langle s \rangle A \times \text{list}\langle s \rangle A$$
$$(\text{++}) \equiv \dots : \Pi A. \text{list}\langle s \rangle A \rightarrow \text{list}\langle r \rangle A \rightarrow \text{list}\langle \infty \rangle A$$

fix **qsort** : list A → list A :=

λx : list A. case x^{list⟨ẑ⟩} of

| nil ⇒ nil

| cons h t^{list⟨z⟩} ⇒ let (s, g) = filter (< h) t^{list⟨z⟩} in
(**qsort** s^{list⟨z⟩}) ++ (cons h (**qsort** g^{list⟨z⟩}))

Example: quicksort

Non-structural recursion

$$\text{filter} \equiv \dots : \Pi A. (A \rightarrow \text{bool}) \rightarrow \text{list}\langle s \rangle A \rightarrow \text{list}\langle s \rangle A \times \text{list}\langle s \rangle A$$
$$(\text{++}) \equiv \dots : \Pi A. \text{list}\langle s \rangle A \rightarrow \text{list}\langle r \rangle A \rightarrow \text{list}\langle \infty \rangle A$$

fix **qsort** : list $A \rightarrow$ list $A :=$
 $\lambda x : \text{list } A. \text{ case } x^{\text{list}\langle \hat{\tau} \rangle}$ of

- | nil \Rightarrow nil
- | cons $h\ t^{\text{list}\langle \tau \rangle} \Rightarrow \text{let } (s, g) = \text{filter } (\lambda h) t^{\text{list}\langle \tau \rangle}$ in
 $(\text{qsort } s^{\text{list}\langle \tau \rangle}) \text{ ++ } (\text{cons } h (\text{qsort } g^{\text{list}\langle \tau \rangle}))$

: $\Pi A. \text{list}\langle s \rangle A \rightarrow \text{list}\langle \infty \rangle A$

Type-based termination

- Handle higher-order data

$$\text{node} : \Pi A.A \rightarrow \text{list} \langle \infty \rangle (\text{rose} \langle s \rangle A) \rightarrow \text{rose} \langle \hat{s} \rangle A$$

- Advantages over syntactic criteria
 - ▶ Expressiveness
 - ▶ Compositional
 - ▶ Easier to understand (specially for ill-typed terms)
 - ▶ Easier to implement (as shown in prototype implementations)
 - ▶ Easier to study (semantically intuitive)
 - ▶ Not intrusive for the user (minimal annotations required)
- Good candidate to replace syntactic criterion in Coq

Coinductive Types

- Coinductive types are used to model and reason about infinite data and infinite processes.
- Coinductive types can be seen as the dual of inductive types.

Inductive types	Coinductive types
Induction	Coinduction
Recursive functions consume data	Corecursive functions produce data

Coinductive Types in Coq

- Streams:

CoInductive stream A := scon : $A \rightarrow \text{stream } A \rightarrow \text{stream } A$

Coinductive Types in Coq

- Stream Empty as an inductive type

CoInductive stream A := scons : A → stream A → stream A

Coinductive Types in Coq

- Streams:

CoInductive stream $A :=$ scon : $A \rightarrow \text{stream } A \rightarrow \text{stream } A$

- Corecursive functions produce streams:

zeroes := cofix $Z :=$ scon(0, Z)

zeroes produce the stream:

scon(0, scon(0, scon(0, ...)))

Coinductive types

Inductive types	Coinductive types
Termination	Productivity

- In proof assistants, termination of recursive functions is essential to ensure logical consistency and decidability of type checking.
- For corecursive functions, the dual condition to termination is **productivity**.
- In the case of streams, productivity means that we can compute any element of the stream in finite time:

cofix $Z_1 := \text{scons}(0, Z_1)$

cofix $Z_2 := \text{scons}(0, \text{tail } Z_2)$

Coinductive types

<u>Inductive types</u>	<u>Coinductive types</u>
Termination	Productivity

- In proof assistants, termination of recursive functions is essential to ensure logical consistency and decidability of type checking.
- For corecursive functions, the dual condition to termination is **productivity**.
- In the case of streams, productivity means that we can compute any element of the stream in finite time:

cofix $Z_1 := \text{scons}(0, Z_1)$ ✓

cofix $Z_2 := \text{scons}(0, \text{tail } Z_2)$

Coinductive types

<u>Inductive types</u>	<u>Coinductive types</u>
Termination	Productivity

- In proof assistants, termination of recursive functions is essential to ensure logical consistency and decidability of type checking.
- For corecursive functions, the dual condition to termination is **productivity**.
- In the case of streams, productivity means that we can compute any element of the stream in finite time:

cofix $Z_1 := \text{scons}(0, Z_1)$ ✓

cofix $Z_2 := \text{scons}(0, \text{tail } Z_2)$ ✗

Coinductive types

<u>Inductive types</u>	<u>Coinductive types</u>
Termination	Productivity

- In proof assistants, termination of recursive functions is essential to ensure logical consistency and decidability of type checking.
- For corecursive functions, the dual condition to termination is **productivity**.
- In the case of streams, productivity means that we can compute any element of the stream in finite time:

cofix $Z_1 := \text{scons}(0, Z_1)$ ✓

cofix $Z_2 := \text{scons}(0, \text{tail } Z_2)$ ✗

(tail Z_2) loops

Syntactic-Based Methods for Productivity

Inductive types	Coinductive types
Termination	Productivity
Guarded-by-Destructor	Guarded-by-Constructor

- Guarded-by-constructor: every corecursive call is performed directly under a constructor
- Same limitations as in the inductive case

Syntactic-Based Methods for Productivity

Inductive types	Coinductive types
Termination	Productivity
Guarded-by-Destructor	Guarded-by-Constructor

- Guarded-by-constructor: every corecursive call is performed directly under a constructor
- Same limitations as in the inductive case

`nats := cofix nats := λn. scon(n, nats (1 + n))`

Syntactic-Based Methods for Productivity

Inductive types	Coinductive types
Termination	Productivity
Guarded-by-Destructor	Guarded-by-Constructor

- Guarded-by-constructor: every corecursive call is performed directly under a constructor
- Same limitations as in the inductive case

`nats := cofix nats := λn. scon(n, nats (1 + n))` ✓

Syntactic-Based Methods for Productivity

Inductive types	Coinductive types
Termination	Productivity
Guarded-by-Destructor	Guarded-by-Constructor

- Guarded-by-constructor: every corecursive call is performed directly under a constructor
- Same limitations as in the inductive case

`nats := cofix nats := λn. scon(n, nats (1 + n))` ✓

`nats := λn. cofix nats := scon(n, map (λx. 1+x) nats)`

Syntactic-Based Methods for Productivity

Inductive types	Coinductive types
Termination	Productivity
Guarded-by-Destructor	Guarded-by-Constructor

- Guarded-by-constructor: every corecursive call is performed directly under a constructor
- Same limitations as in the inductive case

`nats := cofix nats := λn. scon(n, nats (1 + n))` ✓

`nats := λn. cofix nats := scon(n, map (λx. 1+x) nats)`

Syntactic-Based Methods for Productivity

Inductive types	Coinductive types
Termination	Productivity
Guarded-by-Destructor	Guarded-by-Constructor

- Guarded-by-constructor: every corecursive call is performed directly under a constructor
- Same limitations as in the inductive case

`nats := cofix nats := λn. scon(n, nats (1 + n))` ✓

`nats := λn. cofix nats := scon(n, map (λx. 1+x) nats)` ✗

Type-Based Methods for Productivity

- Sized types can be applied to productivity checking as well!

Type-Based Methods for Productivity

- Sized types can be applied to productivity checking as well!
- Dual meaning of size annotations on coinductive types

$\text{stream}\langle s \rangle A$

is the type of streams of which **at least** s elements can be produced

Type-Based Methods for Productivity

- Sized types can be applied to productivity checking as well!
- Dual meaning of size annotations on coinductive types

$\text{stream}\langle s \rangle A$

is the type of streams of which **at least** s elements can be produced

- Size annotations are contra-variant:

$$\frac{r \sqsubseteq s}{\text{stream}\langle s \rangle T \leq \text{stream}\langle r \rangle T}$$

Type-Based Methods for Productivity

- Sized types can be applied to productivity checking as well!
- Dual meaning of size annotations on coinductive types

$\text{stream}\langle s \rangle A$

is the type of streams of which **at least** s elements can be produced

- Size annotations are contra-variant:

$$\frac{r \sqsubseteq s}{\text{stream}\langle s \rangle T \leq \text{stream}\langle r \rangle T}$$

$$\frac{s \sqsubseteq r}{\text{list}\langle s \rangle T \leq \text{list}\langle r \rangle T}$$

Type-Based Methods for Productivity

- Typing rules are similar to the inductive case
- Rules for constructors:

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : \text{stream}\langle s \rangle A}{\Gamma \vdash \text{scons}(M, N) : \text{stream}\langle \hat{s} \rangle A}$$

- Cofixpoint definition is also similar to fixpoint definition:

$$\frac{\Gamma(f : \text{stream}\langle i \rangle A) \vdash M : \text{stream}\langle \hat{i} \rangle A}{\Gamma \vdash \text{cofix } f := M : \text{stream}\langle s \rangle A} \quad i \text{ fresh}$$

Type-Based Methods for Productivity

- Typing rules are similar to the inductive case
- Rules for constructors:

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : \text{stream}\langle s \rangle A}{\Gamma \vdash \text{scons}(M, N) : \text{stream}\langle \widehat{s} \rangle A} \quad \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : \text{list}\langle s \rangle A}{\Gamma \vdash \text{cons}(M, N) : \text{list}\langle \widehat{s} \rangle A}$$

- Cofixpoint definition is also similar to fixpoint definition:

$$\frac{\Gamma(f : \text{stream}\langle i \rangle A) \vdash M : \text{stream}\langle \widehat{i} \rangle A}{\Gamma \vdash \text{cofix } f := M : \text{stream}\langle s \rangle A} \quad i \text{ fresh}$$

Type-Based Methods for Productivity

- Typing rules are similar to the inductive case
- Rules for constructors:

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : \text{stream}\langle s \rangle A}{\Gamma \vdash \text{scons}(M, N) : \text{stream}\langle \widehat{s} \rangle A} \quad \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : \text{list}\langle s \rangle A}{\Gamma \vdash \text{cons}(M, N) : \text{list}\langle \widehat{s} \rangle A}$$

- Cofixpoint definition is also similar to fixpoint definition:

$$\frac{\Gamma(f : \text{stream}\langle i \rangle A) \vdash M : \text{stream}\langle \widehat{i} \rangle A}{\Gamma \vdash \text{cofix } f := M : \text{stream}\langle s \rangle A} \quad i \text{ fresh}$$

$$\frac{\Gamma(f : \text{list}\langle i \rangle A \rightarrow U) \vdash M : \text{list}\langle \widehat{i} \rangle A \rightarrow U}{\Gamma \vdash \text{fix } f := M : \text{list}\langle s \rangle A \rightarrow U} \quad i \text{ fresh}$$

Co-recursive definitions

Examples

$\text{map} : (A \rightarrow B) \rightarrow \text{stream } A \rightarrow \text{stream } B$

$\text{merge} : \text{stream } \text{nat} \rightarrow \text{stream } \text{nat} \rightarrow \text{stream } \text{nat}$

$\text{merge } (1\ 3\ 5\ \dots) (2\ 4\ 6\ \dots) = (1\ 2\ 3\ 4\ \dots)$

$\text{ham} := \text{cofix } \text{ham} : \text{stream } \text{nat} :=$

$\text{scons}(1, \text{merge } (\text{map } (\lambda x. 2*x) \text{ham})$
 $\text{(merge } (\text{map } (\lambda x. 3*x) \text{ham})$
 $\text{(map } (\lambda x. 5*x) \text{ham})))$

$\text{ham} = (1\ 2\ 3\ 4\ 5\ 6\ 8\ 9\ 10\ 12\ 15\ \dots)$

Co-recursive definitions

Examples

$\text{map} : (A \rightarrow B) \rightarrow \text{stream} \langle s \rangle A \rightarrow \text{stream} \langle s \rangle B$

$\text{merge} : \text{stream} \langle s \rangle \text{nat} \rightarrow \text{stream} \langle s \rangle \text{nat} \rightarrow \text{stream} \langle s \rangle \text{nat}$

$\text{merge} (1\ 3\ 5\ \dots) (2\ 4\ 6\ \dots) = (1\ 2\ 3\ 4\ \dots)$

$\text{ham} := \text{cofix } \text{ham} : \text{stream } \text{nat} :=$

$\text{scons}(1, \text{merge} (\text{map } (\lambda x. 2 * x) \text{ham})$
 $\text{(merge } (\text{map } (\lambda x. 3 * x) \text{ham})$
 $\text{(map } (\lambda x. 5 * x) \text{ham})))$

$\text{ham} = (1\ 2\ 3\ 4\ 5\ 6\ 8\ 9\ 10\ 12\ 15\ \dots)$

Co-recursive definitions

Examples

$\text{map} : (A \rightarrow B) \rightarrow \text{stream}\langle s \rangle A \rightarrow \text{stream}\langle s \rangle B$

$\text{merge} : \text{stream}\langle s \rangle \text{nat} \rightarrow \text{stream}\langle s \rangle \text{nat} \rightarrow \text{stream}\langle s \rangle \text{nat}$

$\text{merge} (1\ 3\ 5\ \dots) (2\ 4\ 6\ \dots) = (1\ 2\ 3\ 4\ \dots)$

$\text{ham} := \text{cofix ham} : \text{stream nat} :=$

$\text{scons}(1, \text{merge} (\text{map} (\lambda x. 2*x) \text{ham}^{\text{stream}\langle z \rangle})$
 $\quad (\text{merge} (\text{map} (\lambda x. 3*x) \text{ham}^{\text{stream}\langle z \rangle})$
 $\quad\quad (\text{map} (\lambda x. 5*x) \text{ham}^{\text{stream}\langle z \rangle})))$

$\text{ham} = (1\ 2\ 3\ 4\ 5\ 6\ 8\ 9\ 10\ 12\ 15\ \dots)$

Sized types for coinduction

- Type-based productivity has several advantages over syntactic-based
 - ▶ More expressive
 - ▶ Compositional
 - ▶ Easier to understand (specially for ill-typed terms)
 - ▶ Easier to implement (as shown in prototype implementations)
 - ▶ Easier to study (semantically intuitive)
 - ▶ Not intrusive for the user (minimal annotations required)
- Furthermore, sized types treat inductive and co-inductive types in a similar way

What's next?

What's next?

- Design a type-based termination system for Coq
- Implementation!

What's next?

- Design a type-based termination system for Coq
- Implementation!
- Sombrero line (Barthe et al.) : $\hat{\lambda}$, \hat{F} , \hat{CIC}
- Sizes are declared implicitly (not first class):
- Size inference: little burden for the user
 - ▶ Constraint-based algorithm
 - ▶ Treats fixpoints and co-fixpoints in the same way
- Still some issues remain in order to adapt to full Coq

In a future Coq version ...

```
Fixpoint map ι (f : A -> B) (xs : List<ι> A) : List<ι> B :=  
  match xs with  
  | nil      => nil  
  | cons h t => cons (f h) (map f t)  
end.
```

In a future Coq version ...

```
Fixpoint map ι (f : A -> B) (xs : List<ι> A) : List<ι> B :=
  match xs with
  | nil       => nil
  | cons h t => cons (f h) (map f t)
  end.
```

Check map.

```
map : ∀ ι. (A -> B) -> List<ι> A -> List<ι> B.
```

In a future Coq version ...

```
Fixpoint map  $\iota$  (f : A -> B) (xs : List< $\iota$ > A) : List< $\iota$ > B :=  
  match xs with  
  nil      => nil  
  cons h t => cons (f h) (map f t)  
end.
```

Check map.

```
map :  $\forall \iota. (A \rightarrow B) \rightarrow \text{List}<\iota> A \rightarrow \text{List}<\iota> B.$ 
```

```
Fixpoint ntail  $\iota$  A (x : nat< $\iota$ >) : List A  $\rightarrow$  List A :=  
  ...
```

In a future Coq version ...

```
Fixpoint map ι (f : A -> B) (xs : List<ι> A) : List<ι> B :=
  match xs with
  | nil       => nil
  | cons h t => cons (f h) (map f t)
  end.
```

Check map.

```
map : ∀ ι. (A -> B) -> List<ι> A -> List<ι> B.
```

```
Fixpoint ntail ι A (x : nat<ι>) : List A → List A :=
  ...
```

Check ntail.

In a future Coq version ...

```
Fixpoint map  $\iota$  (f : A -> B) (xs : List< $\iota$ > A) : List< $\iota$ > B :=
  match xs with
  nil      => nil
  cons h t => cons (f h) (map f t)
end.
```

Check map.

```
map :  $\forall \iota$ . (A -> B) -> List< $\iota$ > A -> List< $\iota$ > B.
```

```
Fixpoint ntail  $\iota$  A (x : nat< $\iota$ >) : List A  $\rightarrow$  List A :=
  ...
```

Check ntail.

```
ntail :  $\forall \iota \forall j$ . forall A, nat< $\iota$ > -> List< $j$ > A -> List< $j$ > A.
```


In a future Coq version ...

```
Fixpoint map  $\iota$  (f : A -> B) (xs : List< $\iota$ > A) : List< $\iota$ > B :=  
  match xs with  
  nil      => nil  
  cons h t => cons (f h) (map f t)  
end.
```

Check map.

```
map :  $\forall \iota. (A \rightarrow B) \rightarrow \text{List}<\iota> A \rightarrow \text{List}<\iota> B.$ 
```

```
Fixpoint ntail  $\iota$  A (x : nat< $\iota$ >) : List A  $\rightarrow$  List A :=  
  ...
```

Check ntail.

```
ntail :  $\forall \iota \forall J_1 \forall J_2. J_2 \sqsubseteq J_1 \Rightarrow$   
  forall A, nat< $\iota$ > -> List< $J_1$ > A -> List< $J_2$ > A.
```

Summary

- Keep extending the guard condition is not sustainable
- Time is right to rethink termination checking in Coq
- Sized types seem to be an ideal candidate
 - ▶ More expressive
 - ▶ Compositional
 - ▶ Easier to study and implement

Summary

- Keep extending the guard condition is not sustainable
- Time is right to rethink termination checking in Coq
- Sized types seem to be an ideal candidate
 - ▶ More expressive
 - ▶ Compositional
 - ▶ Easier to study and implement
- Project to start at CMU-Q in September
 - ▶ Careful design before implementation

Summary

- Keep extending the guard condition is not sustainable
- Time is right to rethink termination checking in Coq
- Sized types seem to be an ideal candidate
 - ▶ More expressive
 - ▶ Compositional
 - ▶ Easier to study and implement
- Project to start at CMU-Q in September
 - ▶ Careful design before implementation
- Is this an opportunity to rethink coinduction in Coq?

Summary

- Keep extending the guard condition is not sustainable
- Time is right to rethink termination checking in Coq
- Sized types seem to be an ideal candidate
 - ▶ More expressive
 - ▶ Compositional
 - ▶ Easier to study and implement
- Project to start at CMU-Q in September
 - ▶ Careful design before implementation
- Is this an opportunity to rethink coinduction in Coq?

Thank you!